LBL-22033

CABLE: A Language Based on the Entity-Relationship Model

Arie Shoshani

Computer Science Research Department
University of California
Lawrence Berkeley Laboratory
Berkeley, California 94720

January, 1978

This research was supported by the Applied Mathematics Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under contract DE-AC03-76SF00098.

CABLE: A Language Based on the Entity-Relationship Model

by
Arie Shoshani
Lawrence Berkeley Laboratory

I. Introduction

Over the last several years the concepts of "data independence" and "user-oriented" data languages have been emphasized in the data management area. Briefly, these concepts refer to the separation of the physical organization of a database from its logical structure, and the development of data models and languages based only on some logical model. The best known model that emphasized these concepts is the relational model [1] and the different languages developed upon it such as SEQUEL [2] or QUEL [3].

The introduction of the relational model triggered a controversy as to its merit relative to the CODASLY network model [4] and more traditional hierarchical models (e.g. [5]). The issues are based on how "natural" the model is, how easy it is to specify a request, and whether the model promotes "non-procedurality" in the languages based upon it. Roughly speaking, a language is considered non-procedural if it has facilities to specify **what** is wanted, rather than **how** to get it. We will discuss some of these issues in the next section and argue that the Entity-Relationship model [6] provides a good basis for "natural" user languages. The Entity-Relationship model was introduced in an attempt to unify the different approaches. It includes a great deal of semantics not present in the relational model.

In this paper we discuss a data language, called CABLE (**c**h**a**in-**b**ased **l**anguag**e**), that is designed to take advantage of the semantics present in the Entity-Relationship model (the use of the term chain will become clear as we proceed). In the next section we discuss the issue of what is a natural model and the concept of procedurality at the logical level, which leads to the notion of chains. In the following section we discuss the main features of the CABLE language by way of examples and show how chains can be used to represent many desirable constructs.

II. On "Naturalness" and Non-Procedurality

The issue of whether a model is natural cannot be resolved at the level of "what do most people like best." The preference that an individual has for a data model depends not only on the application's data, but also on personal biases and previous experience. Further the difficulty of using languages based on the models seems to vary with the application [7]. We would propose a criteria for what is natural based on the use of natural languages (English in our case) as follows: A data language is considered **natural** if it requires no more parameters and constructs than would be required when using ordinary English. A model is natural if it can support a data language that is natural.

consider the following example: "Find all employees working for the manager Jones." Assume that a relational model of the database contains the relations:

```
DEPT (NAME, FLOOR, MGR)
EMPL (NAME, SAL, DEP)
```

In SEQUEL [2] the query would be:

```
SELECT NAME
FROM EMP
WHERE DEP =
        SELECT NAME
        FROM DEPT
        WHERE MGR = 'JONES'
```

and its equivalent in QUEL for the INGRES system [3].

```
RANGE OF D IS DEPT
RANGE OF E IS EMPL
RETRIEVE (E. NAME)
WHERE D. MGR = 'JONES' AND
   E. DEP = D. NAME
```

It will be more natural according to the definition above to use something like

```
SELECT EMPL. NAME
WHERE MGR = 'JONES'
```

This query has the parameters "EMPL. NAME" which stems from "employee" in the English query, and MGR = 'JONES' which stems from "manager Jones." In the QUEL query another item was required: E. DEP = D. NAME, which associates explicitly the two relations. Similarly, the SEQUEL query required the additional indented

SELECT to do the same. The reason for this requirement stems from the relational model, in which relationships are not specified explicitly. As a result all association must be specified in the relational language. We claim that if relationships are known, the user should not be required to specify them in the data language. In the English query the user did not express this association since it was part of his model that employees work in departments. In the Entity-Relationship model such associations can be made explicit and therefore a language taking advantage of this semantic knowledge can be developed. In the next sections we will discuss how this is done in CABLE.

Note that in the previous example we used MGR without mentioning that it belongs to the relation DEPT. This is only possible if attribute names are unique. Indeed, this condition holds true for relationships as well, and we will later discuss how we deal with it.

The property of non-procedurality was discussed much in the context of comparing data languages based on the relational model and the CODASYL Data Manipulation Language (DML) [4] which is based on a network model. However, the detailed procedurality of this DML stems from it being a low-level record at-a-time manipulation language. A user of this language is expected to know about the physical organization of the database (for example, in terms of pointer chains) and use this knowledge to move between records. The languages based on the relational model are at a logical level and therefor this comparison is not useful for the purpose of understanding non-procedurality. It is more appropriate to ask what is procedurality at the logical level.

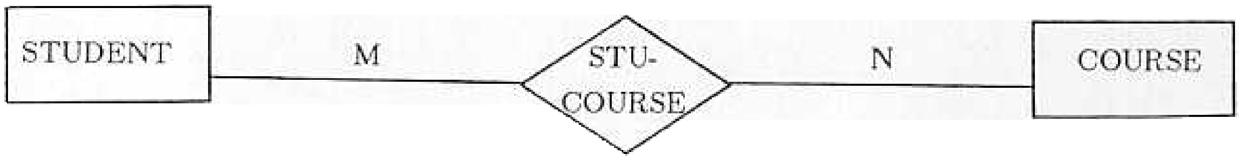
Consider the following query: "what are the parts used in projects of the electronic department." In this query we actually specify how the information is to be accessed; i.e. start with the electronics department, find all projects in this department, then find all parts used by these projects. If we were to specify only what we want, we would say "what are the parts used in the electronics department" not mentioning projects. Thus, the first query can be considered procedural, at least relative to the second one. An example of a language that is procedural at the logical level is LSL, described in [8]. In the use of natural languages, people would use the more procedural specification when they are less familiar with the database or when there may be more than one interpretation to the abbreviated form.

We believe that procedurality at the logical level is the detailed specification of a chain through entities and relationships of the database. the less elements are specified in the chain, the less procedural the query specification is. Further, how procedural a specification would be depends on the user's view of the logical structure of the database, and on his experience with it. Procedurality is a "natural" feature in the sense that it exists in natural language. It is a valid feature to exist in data languages, but should be required only if ambiguities arise. The concept of chains representing procedurality at the logical level are the basis for the language CABLE.

III. Features of CABLE

As mentioned earlier CABLE is designed to take advantage of semantic knowledge provided by the Entity-Relationship model. At the same time, it preserves an important feature of relational languages: the ability to relate entities by value association. We will proceed by discussing features of the language and illustrating them with examples. But first, we need to introduce briefly the basic concepts of the Entity-Relationship model.

"Entities" represent "things" in the databases, such as people, organizations or events. "Attributes" represent information about entities, such as age and salary of a person. "Relationships" represent association between entities, such as the fact that employees are assigned to departments. Relationships can exist between two or more entities and specify whether the association is one-on one, one-to many or many-to-many. In most cases entities have an independent existence (i.e. they would exist in the database regardless of whether other entities or relationships exist), while attributes and relationships are always dependent. There are exceptions to the independence of entities; for example, in a database about employees and children, children are entities whose existence can be defined to be dependent on employee. A key feature of the model is that relationships can have attributes which represent information about the combination of entities involved in the relationship. For example, in a database where students and courses are entities, a relationship student-course can have an attribute grade. More details about the model can be found in Chen [6]. Chen uses a diagrammatic notion where rectangles represent entities and diamonds represent relationships. Attributes are not represented so that the diagram does not get too busy. Thus the last example would be diagrammed as follows:



We will use the same notation for our examples below and will refer to the following database in figure 1 (part of the diagram in [6]) for examples in the rest of the paper.

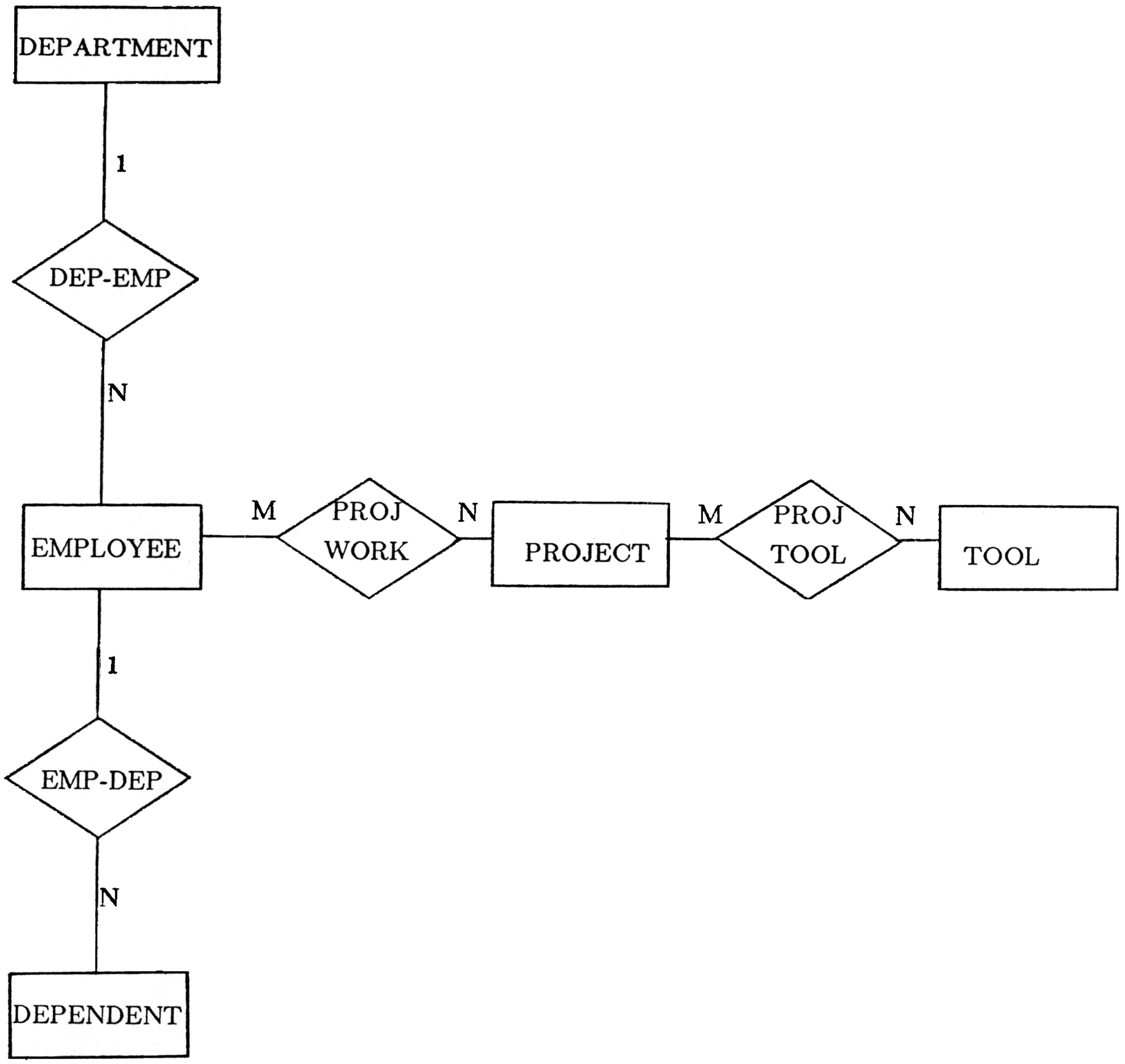


FIGURE 1: An Entity-Relationship Model Example

The entities have the following attributes:

DEPARTMENT (NAME, BUILDING)
EMPLOYEE (NAME, AGE, SALARY)
DEPENDENT (NAME, AGE)
PROJECT (NUMBER, FLOOR)
TOOL (NUMBER, QUANTITY)

In addition the relationship PROJ. WORK (which is many-to-many) has an attribute TIME designating the portion of the employee's time spent on a given project.

We can now proceed to discuss the features of the CABLE language. The basic selection elements are chains. Chains represent selection paths through the entities and relationships of the database. Chains are made of beads, which are an elementary selection criteria. The following SELECT statement is an example of a chain:

E1 find all projects which employ people from departments in building 5.
$\underline{\text{SELECT}}$ PROJECT/EMPLOYEE/DEPARTMENT. BUILDING = '5'
where SELECT is a keywork (underlined), beads of the chain are separated by '/' and '.' stands between an entity and one of its attributes. Note that beads may be made of entity names only.

E1 represents only a selection criteria. To output elements one adds an OUTPUT statement. The output can refer to any entities in the chain. For example suppose that for E1 we want the following output:

E2 output the project numbers and floors as well as department names. The corresponding OUTPUT statement will simply be:
$\underline{\text{OUTPUT}}$ PROJECT. NUMBER, PROJECT. FLOOR, DEPARTMENT. NAME

Chains do not have to include all elements on their path as long as no ambiguity arises (We will discuss what can be done in such cases in section 4). In addition, attributes do not have to be preceded by their entities if they have a uniques name in the database. Thus the above request could be represented in abbreviated form as follows (including E1 and E2)

E3: OUTPUT PROJECT. NUMBER, FLOOR, DEPARTMENT. NAME
SELECT BUILDING = '5'

In E3 all missing parameters can be added by the system. In such cases, the complete statement can be optionally shown to the user before proceeding to perform the request.

Note that in the previous examples no relationships were specified, relying on the knowledge of their existence. E3 represents the minimum number of parameters and constructs necessary to represent this request unambiguously.

There is no distinction in the language between entities or relationship as elements of beads. This makes sense, since relationships can have attributes that we may want to include in the qualification criteria. Before showing such an example we should describe in more detail the composition of a bead. There is one bead for each entity or relationship involved in the chain. Beads can have qualification criteria to be applied to its entity. A bead may contain no qualification criteria, just an entity or relationship name. An example of a qualification bead is:

EMPLOYEE. AGE > 35 AND SALARY < 10000

Bead qualification can be applied anywhere along the chain as in the following example:

E4 what projects have employees over 65 years old from the R&D department
OUTPUT PROJECT. NUMBER
SELECT EMPLOYEE. Age > 65/DEPARTMENT. NAME = 'R&D'

An example where a relationship and its attribute are involved would be:

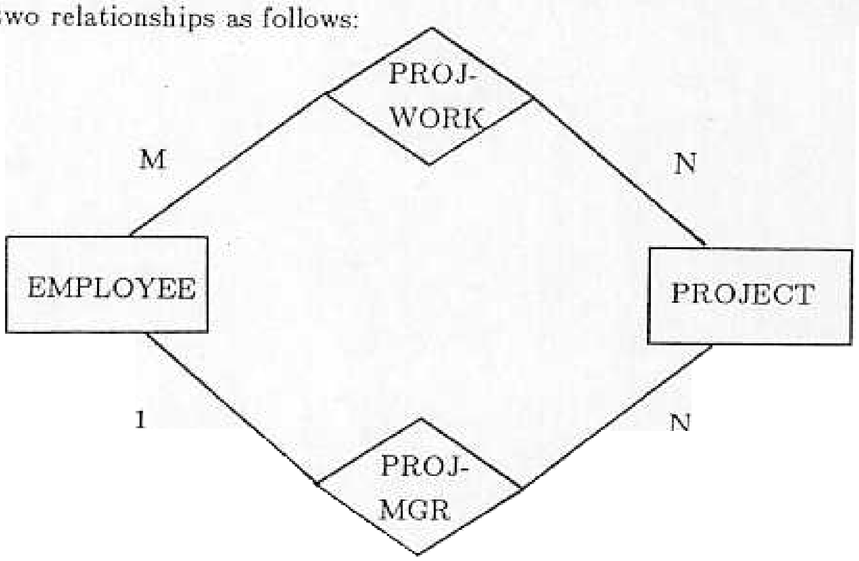
E5: What employees work in project number 7 more than half-time.
OUTPUT EMPLOYEE. NAME
SELECT PROJ-WORK. TIME > 0.5/PROJECT. NUMBER = 7

Chains can be connected by the logical operators AND/OR. It is often the case that two or more conditions represented with an AND are equivalent to a chain. For example E5 could be represented as two chains each having one bead as follows:

E6: <u>OUTPUT</u> EMPLOYEE. NAME
SELECT PROJ-WORK. TIME > 0.5 <u>AND</u> PROJECT. NUMBER = 7

This equivalence can easily be detected and both forms are permissible depending on the user's view.

Beads in chains can be used effectively to choose between two relationships existing between entities. For this purpose we will add a relationship between EMPLOYEE and PROJECT that indicates which employees manage projects. We not have two relationships as follows:



We can choose to use one of the relationships by selecting the appropriate bead as in the following example:

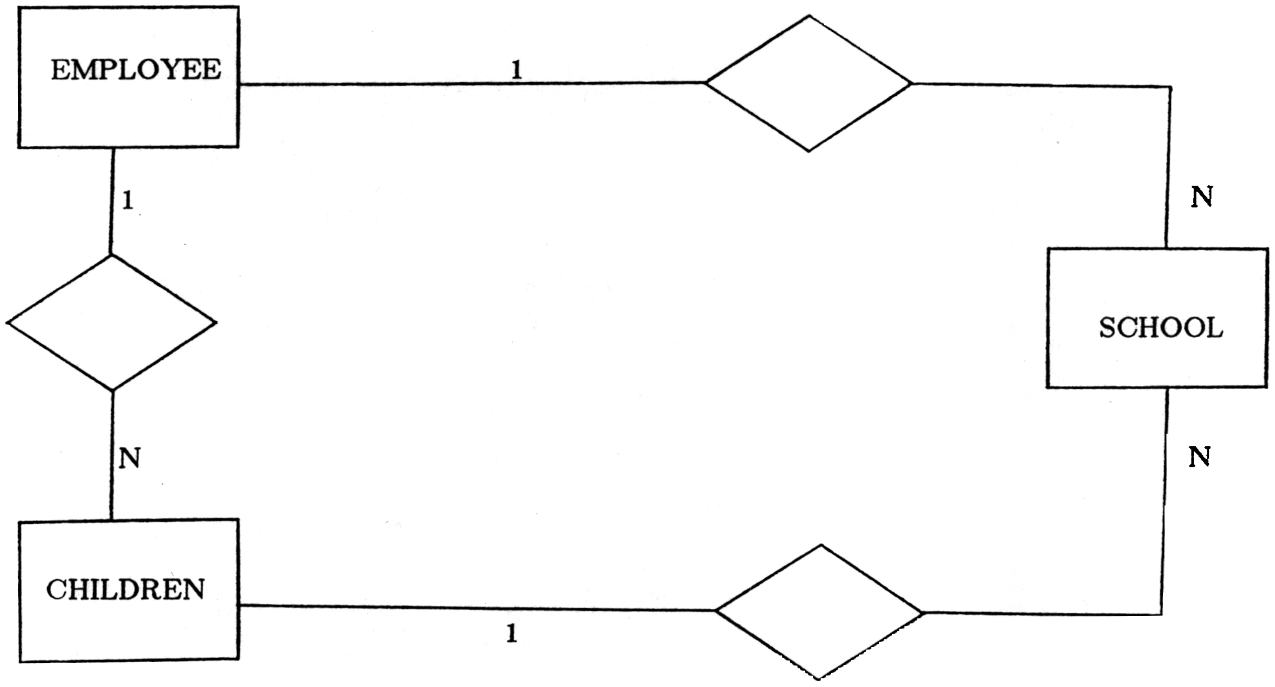
E7 Who is the manager of project 17?
OUTPUT EMPLOYEE. NAME
<u>SELECT</u> PROJ-MGR/PROJECT. NAME = 17

Entities and relationships can be traversed more than once using chains. For Example (refer again to Figure 1):

E8: What projects use the tools used by project 5?

<u>OUTPUT</u> EMPLOYEE. NAME
<u>SELECT</u> PROJECT/TOOL/PROJECT. NUMBER = 5.

One of the more difficult conditions to express in a data language is the condition caused by ellipsis in natural languages, such as by the words their, its etc. For example: Who are the employees that went to the same school as their children? This usually requires the designation of a variable that is used in two parts of the query. In CABLE this is done by putting the condition between two chains. To illustrate the point consider the following diagram:

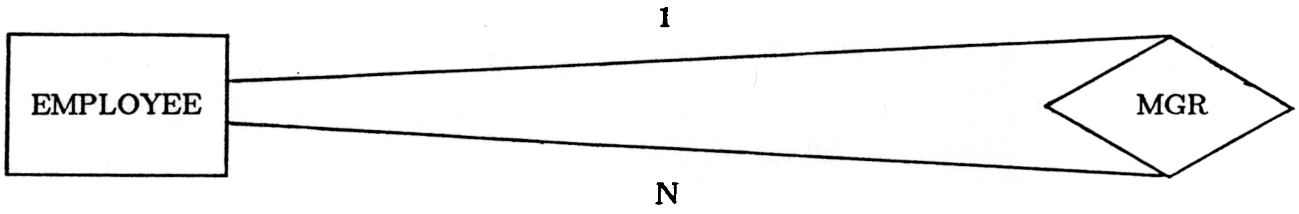


For the example above the query is expressed by equating two chains as follows

E9: Who are the employees that went to the same school as their children?

<u>OUTPUT</u> EMPLOYEE. NAME
<u>SELECT</u> [SCHOOL. NAME/EMPLOYEE] = [SCHOOL. NAME/CHILDREN/EMPLOYEE]
The term EMPLOYEE at the end of both chains serves as the linking variable.

The Entity-Relationship model allows relationships to be recursive, i.e. defined on a single entity as in the example below:



In such a case, ~~MSR~~ MGR is considered not only a relationship name, but also another name for the entity EMPLOYEE. Thus the following example can be simply expressed as follows:

E10: What is the salary of the manager of Jones?
OUTPUT MGR. SALARY
SELEDCT EMPLOYEE. NAME = 'JONES'
The system can create two references for EMPLOYEE and use the relationship MGR to express it internally as an explicit chain:
OUTPUT EMPLOYEE 1. SALARY
SELECT EMPLOYEE 1/MGR/EMPLOYEE 2 = 'JONES'
Similarly the next famous example can be expressed simply as follows:

E11: Who are the employees that earn more than their managers?
OUTPUT EMPLOYEE. NAME
SELECT EMPLOYEE. SALARY > ~~MSG~~ MGR. SALARY

Finally, we discuss the ability to associate entities explicitly, which is an important feature in relational languages. In CABLE this can be done by using a bead in a chain for that purpose. this feature is similar to the LINK capability in LSL [8]. The bead is composed of the keyword LINK followed by an association criteria. For example, suppose that we have a separate entity of houses where one of its attributes is owner. Then the following query can be expressed using the LINK bead.

E12: What is the salary of the employee who owns house NUMBER 179?
OUTPUT EMPLOYEE. SALARY
SELECT LINK EMPLOYEE. NAME = HOUSE. OWNER/HOUSE. NUMBER = 179
The LINK bead is useful for entities that are not related when we wish to associate them by value, but can also be used between entities that are already related.

IV. Dealing with Ambiguities

The entity-relationship model represents a general network or graph. Therefore, it is possible that if chains are not fully specified, multiple choices arise. In such cases there is no way of knowing the user's intention and he must be consulted. It seems intuitive that for most of the cases the choice of the shortest path would be a good guess, but this assumption will require substantiation. There are several possible approaches on how to represent the response to a user in the case of multiple chains. assuming that the user is an interactive "casual" user, a reasonable technique seems to be to ask him about the possible beads in chains one at a time, and progressively eliminate chains not including the indicated beads. Also, the presentation to the casual user of a chain selected by the system, must be done using English formats similar to the way it is done in the Rendezvous system [9]. We believe that only a small fraction of queries will require disambiguation since it is unlikely that only the ends of long chains will be specified by users, without their mentioning some beads in between.

V. Conclusion

We have described a language based on the Entity-relationship model, called CABLE. The main feature of this language is that it requires a small number of parameters and constructs in order to specify a query. This is particularly true when compared to relational languages that are based on value association in the queries. In fact, we believe that this language requires in most cases no more parameters and constructs than used in natural language. Therefore, the language is close to the conceptual constructs that (casual) users have. We also believe that the ability to use chains is a natural feature that facilitates the expression of many constructs easily. At the same time an experienced user can choose to express his queries in detailed chains for the expression of complicated conditions.

We have not discussed many features that a data language should have, such as update or modify facilities, since we wanted to concentrate on the concepts of language design. There are many good ideas in the literature that could be used with this approach, such as the definition and creation of LINKs in LSL [8], or the storing of results into temporary structures as is done in relational languages.

VI. References

1. Codd, E.F., *"A Relational Model of Data for Large Shared Data Banks,"* Comm. ACM 13, 1970, pp. 377-387.
2. Chamberlin, D.D., and boyce, R.F., *"SEQUEL: A Structured English Query Language,"* Proc. ACM SIGMOD Workshop, 1974, pp. 249-264.
3. Stonebraker, M.R., Wong, E., Kreps, P., and Held, G., *"The Design and Implementation of INGRES,"* ACM Trans. on Database Systems, Vol. 1, 1976, pp. 189-222.
4. CODASYL Database Task Group Report, 1971, ACM, New York.
5. MRI Systems Corp., SYSTEM 2000 Publications, 1972, Austin, Texas.
6. Chen, P. P-S, *"The Entity-Relationship Model: Toward a Unified View of Data,"* ACM Trans. on Database Systems, Vol. 1, 1976, pp. 9-36.
7. Lockovsky, F.H., and Tsichritzis, D.C., *"User Performance Considerations in DBMS Selection,"* Proc. ACM-SIGMOD Conference, 1977, pp. 128-134.
8. Tsichritzis, D.C., *"LSL: A Link and Selector Language,"* Proc. ACM-SIGMOD Conference, 1976, pp. 123-133.
9. Codd, E.F., *"Seven Steps to Rendezvous with the Casual User,"* in Database Management, Klimbie and Koffeman, ed., pp. 179-199, North-Holland, Amsterdam.